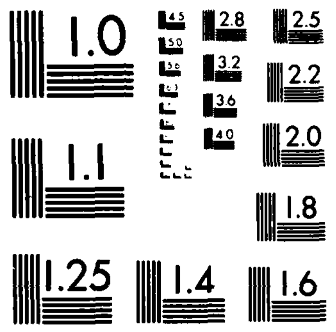
AD-A132 851 AN EFFICIENT GENERATOR OF UNIFORMLY DISTRIBUTED RANDOM DEVIATES BETWEEN Z. (U) WISCONSIN UNIV-MADISON MATHEMATICS RESEARCH CENTER A THESEN AUG 83 1/1

UNCLASSIFIED MRC-TSR-2554 DAAG29-80-C-0041 F/G 12/1 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A132 851

2

MRC Technical Summary Report #2554

AN EFFICIENT GENERATOR OF UNIFORMLY DISTRIBUTED RANDOM DEVIATES BETWEEN ZERO AND ONE

Arne Thesen

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53705

August 1983

(Received May 4, 1983)

Sponsored by

U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

UNIVERSITY OF WISCONSIN-MADISON
MATHEMATICS RESEARCH CENTER

AN EFFICIENT GENERATOR OF UNIFORMLY DISTRIBUTED RANDOM DEVIATES BETWEEN ZERO AND ONE

Arne Thesen*

Technical Summary Report #2554
August 1983

ABSTRACT

We show that a uniformly distributed random deviate between zero and one can be constructed by combining two independent random variables: 1) a random exponent drawn from a geometric distribution, and, 2) a random mantissa drawn from a uniform distribution. Algorithms for generating these random variables are developed, and a Pascal procedure combining these into a computationally efficient random number generator for micro computers is presented. The resulting generator is computationally efficient as it: 1) uses no floating point arithmetic, and, 2) uses on the average only 3.5 random bytes to construct a four byte random deviate. Finally we show empirically that the generator has both an unusually long period and excellent statistical properties.

AMS (MOS) Subject Classifications: 65C10, 68J10, 90-04

Key Words: Random Number Generators, Uniform Distribution, Micro Computers

Work Unit Number 6 (Miscellaneous Topics)

---

*Department of Industrial Engineering and Mathematics Research Center, University of Wisconsin-Madison, Madison, WI 53705

Sponsored by the United States Army under Contract No. DAAG29-80-C-0041.

SIGNIFICANCE AND EXPLANATION

Conventional generators of uniformly distributed deviates between zero and one usually obtain their deviates by converting a potentially large random integer to floating point notation and scaling it down as required. This is a reasonable procedure for a computer with a large wordsize and with floating point arithmetic implemented in hardware. However many micro and mini computers use 16 bit integer arithmetic and have floating point arithmetic implemented in software. For these computers, the conventional approach to generation of uniformly distributed deviates between zero and one is quite undesirable as it is both time consuming and yields deviates with a short period and a low resolution.

In this paper we present an alternative approach. The resulting generator is computationally efficient and has good statistical properties. In addition it has an unusually long period (we stopped measuring at 64,000,000) even when driven by short period (32,768) congruential generators, and like numbers appear at random intervals. The generator is based on the fact that a floating point number is represented in the computer in terms of two variables (a mantissa and an exponent). These are developed independently of each other without the use of floating point arithemtic.

The good performance characteristics of the proposed generator are probably due to the following: (1) no floating point arithmetic is performed, (2) a variable number of random bytes (between three and six) is used to construct a number, (3) half the time the process is extremely simple and requires only three bytes, and, (4) on the average about 3.5 random bytes are required to construct a four byte random floating point number.

AN EFFICIENT GENERATOR OF UNIFC ̵ILY DISTRIBUTED RANDOM DEVIATES BETWEEN ZERO AND ONE

Arne Thesen*

INTRODUCTION

Conventional generators of uniformly distributed deviates between zero and one usually obtain their deviates by converting a potentially large random integer to floating point notation and scaling it down as required. This is a reasonable procedure for a computer with a large wordsize and with floating point arithmetic implemented in hardware. However many micro and mini computers use 16 bit integer arithmetic and have floating point arithmetic implemented in software. For these computers, the conventional approach to generation of uniformly distributed deviates between zero and one is quite undesirable as it is both time consuming and yields deviates with a short period and a low resolution.

In this paper we present an alternative approach. The resulting generator is computationally efficient and has good statistical properties. In addition it has an unusually long period (we stopped measuring at 64,000,000) even when driven by short period (32,768) congruential generators, and like numbers appear at random intervals. The generator is based on the fact that a floating point number is represented in the computer in terms of two variables (a mantissa and an exponent). These are developed independently of each other without the use of floating point arithmetic.

Our implementation of the algorithm is based on the conventions for representation of floating point numbers adopted by the AMD 9511 (OR INTEL 8231A) hardware floating point unit [1]. This is the convention used in most micro computer compilers such as Pascal/MT+ and FORTRAN-80. This convention differs slightly from (but is conceptually identical to) the proposed IEEE standard notation used in many other systems [2]. The decision to use the de facto micro-computer standard rather than the IEEE standard reflects the present unavailability of efficient floating point random number generators on micro computers. An implementation of the algorithm under the IEEE standard should follow our examples quite closely.

The good performance characterisitics of the proposed generator are probably due to the following: (1) no floating point arithmetic is performed . (2) a variable number of random bytes (between three and six) is used to construct a number. (3) half the time the process is extremely simple and requires only three bytes, and, (4) on the average about 3.5 random bytes are required to construct a four byte random floating point number.

FLOATING POINT NOTATION

All floating point conventions are based on the fact that any rational number X can be represented as the product of a fractional part and an integer part. At the cost of not being able to accurately represent all rational numbers. These conventions are implemented as follows:

*Department of Industrial Engineering and Mathematics Research Center, University of Wisconsin-Madison, Madison, WI 53705

Sponsored by the United States Army under Contract No. DAAG29-80-C-0041

$$X = \frac{m}{M} * 2^{e} \qquad \text{Equation 1}$$

where X = The number to be represented

e = an integer exponent

M = a constant (usually a power of two).

m = the mantissa. $m >= M/2$ unless $X = 0$ in which case $m = 0$

The precision of X is determined by M while the range of X is determined by the permissible range of e. Note that the fractional part (m/M) of X is always in the range 1/2 to 1 regardless of the value of X (as long as X is nonzero). In Figure 1 we present the ranges of values taken on by e and m/M for different values of X between zero and one.

X	e	m/M
1.0 >x>=0.5	0	0.5 - 1.0
0.5 >x>=0.25	-1	0.5 - 1.0
0.25 >x>=0.125	-2	0.5 - 1.0
0.125 >x>=0.0625	-3	0.5 - 1.0
0.0625>x>=0.03125	-4	0.5 - 1.0

Figure 1: Values of e and m/M for numbers on [0-1]

It is seen that the exponent e defines the range of a number (i.e. 0.5 > x>=0.25), and that the mantissa m defines the specific value of the number within this range.

RANDOM VARIABLES ON [0-1]

From the preceding discussion, we see that a random variable u on [0-1] can be expressed as a function of a random exponent e and a random mantissa m as follows:

$$u = \frac{m}{M} * 2^{e} \qquad \text{Equation 2}$$

where u = random variable on [0-1]

e = random variable drawn from the distribution:

$$\Pr\{e = i\} = 2^{-(i+1)} \qquad i = 0,1,2,..$$

M = a constant

m = Uniformly distributed random variable on $[M/2, M)$.

Perhaps the most useful consequence of the fact that u is expressed as a function of two independent random variables is the fact that it enables us to propose an algorithm with a period so long that we have been unable to empirically measure it.

As shown in Figure 2, the algorithm starts out by assigning a value of zero to the exponent (defining a number on [0.5-1.0]) and a random value to the mantissa. A check is made to see if the resulting mantissa has a valid value (there is a 50% probability that this is so), if it does, the algorithm stops as a valid number has been produced.

```
 I. INITIAL ASSIGNMENTS

      e  = 0             (Correct value if u > 1/2)
      m  = U(0,M)        (Uniform deviate on 0-M)

II. IS ADDITIONAL WORK REQUIRED?

      If m >= M/2      { this happens half the time}
      then  goto step IV
      else m = m + M/2.

III. ADJUST EXPONENT

    A. Draw random byte(s) until a nonzero byte is found:

      k = RandomByte
      while k = 0    { this happens with a probability of 1/256  }
        e = e - 8
        k = RandomByte.

    B. Scan the byte for the first nonzero bit:

     while k < 128
       k = k  * 2
       e =  e - 1.

IV. RANDOM VARIABLE IS u = f(e,m)
```

Figure 2: Algorithm for generating u

If the mantissa is not valid (i.e. its first bit is not one), then M/2 is added to m to make it valid, and a procedure for generating a random exponent is entered. This procedure is based on the premise that the number of zeroes preceding the first one in a random stream of bits follows the geometric distribution with p =0.5. Random bytes are drawn until a nonzero byte is found. This byte is then scanned until the first nonzero bit is found. The resulting exponent is computed as the negative value of eight times the number of zero valued bytes plus the number of consecutive zero valued bits in the last byte.

Expected Effort

While a fixed number of random bytes are always used to generate a random mantissa, the number of byt[illegible] used to generate a random exponent is itself a random var[illegible]. It can be shown that the distribution of **n** is:

$$Pr\{n = i\} = 1/2 \qquad \text{for } i = 0$$

$$= 255 * (1/256)^{i} /2 \qquad \text{for } i = 1,2,3,...$$

Some of the values of this distribution are shown in Figure 3:

n	0	1	2	3
Pr(n bytes required)	.5000	.4980468	.0019454	.0000075

Figure 3: Number of bytes required to generate **e**.

The expected value of **n** is: E(**n**) = 128 / 255 = 0.5019607

Assuming that one integer multiplication is required to generate one random byte, and that two integer multiplications are required to generate a random mantissa, then an average of 2.5 integer multiplications (and **no** floating point operations) are required per floating point random variable. A more efficient random byte generator may bring this level of effort down even further.

Expected period

The period of the resulting generator is a function of the periods of the generators used to get the random mantissa **m** (**P(m)**) and the random exponent **e** (**P(e)**). If these generators have identical periods, then the random variable **u** will also have this period. However if **P(e)** and **P(m)** are relative prime then the period of **u** is equal to the product of these periods.

P(m) is readily established as a function of the period of the driving byte generator. However **P(e)** is more elusive. Its value appears to depend on the value of the multiplier used in the driving random byte generator. Empirical tests using the programs presented later (Figures 7 and 8) show that **P(e)** is practically unmeasurable for some values (3993, 9237 and 14789) of **a** (the multiplier ,Fig. 8), while for other values (1221,2837,2501,7261 and 12125) it is approximately equal to (but relative prime to) the period of the driving generator.

In both cases it is reasonable to expect that the resulting period of **u** will be very large. Our empirical results bear out this conclusion.

AN IMPLEMENTATION

Notation

The AMD 9511 implementation of floating point notation is shown in Figure 4:

```
 bit 31    30      29 28         25 24 23      16 15        8 7         0
+--------+--------+--+--+---------+--+--+--------+--+--------+--+-------+--+
! sign   ! sign   !   value of    !             Mantissa                   !
!  of    !  of    !  of exponent  !most signif!             !least signif!
!exponent!mantissa!  !  !         !icant byte !             !icant byte  !
+--------+--------+--+--+---------+--+--+--------+--+--------+--+-------+--+
```

Figure 4: AMD 9511 convention for floating point notation

Three bytes are allocated to the mantissa and one byte is used for the exponent and the two sign bits. Note that the first bit of the mantissa (always a one) is explicitly included as bit 24. Since the first two bits of byte 1 are used for sign information, only 6 bits are available for the order of magnitude of the exponent. Exponents are therefore restricted to the range of [-64 to +63].

For numbers between zero and one the sign bit of the mantissa is zero and the sign bit of the exponent is one if the number is less than 0.5 and zero otherwise. The effect of these sign bits and the fact that the first bit of **m** is always 1 is shown in Figure 5.

X	e	m	b1	b2	b3,b4
1.0 >x>=0.5	0	16,777,215-8,388,608	0	128-255	0-255
0.5 >x>=0.25	-1	16,777,215-8,388,608	127	128-255	0-255
0.25 >x>=0.125	-2	16,777,215-8,388,608	126	128-255	0-255
0.125 >x>=0.0625	-3	16,777,215-8,388,608	125	128-255	0-255
0.0625>x>=0.03125	-4	16,777,215-8,388,608	124	128-255	0-255

Figure 5: Values of individual bytes for numbers on [0-1]

The IEEE implementation of this notation is shown in Figure 6.

```
 bit 31  30  29          23  22  21  20                  2   1   0
+--------+---+-----------+---+---+---+-------------------+---+---+---+
! Sign   ! Biased exponent !  Unsigned mantissa with most          !
!  of    ! The bias is 127 !   significant bit missing             !
!Mantissa!  !   !        !  !   !   !                  !   !   !   !
+--------+---+-----------+---+---+---+-------------------+---+---+---+
```

Figure 6: Proposed IEEE standard for Floating Point Notation.

It is seen that the main differences between this notation and the AMD9511 notation are the facts that (1) the most significant bit of the mantissa (always a one) is not included in the IEEE standard while it is in the AMD9511 notation, (2) the IEEE standard allocates one more bit to the exponent, thus expanding the range of values that can be represented, and (3) the exponent-mantissa boundary is no longer at a byte boundary. None of these differences introduce serious implementation problems. Note however, that the logical and physical order of bytes in a variable are not always the same. For example in many systems the least significant byte of an integer is stored below the more significant one. This may cause the exponent to appear as the second rather than the first byte of the variable.

A Pascal Procedure

The main difficulty in implementing the proposed algorithm in Pascal is to get access to the individual bits of a real variable. As shown in the program listing in Figure 7, we obtain this access through the use of a variant record that defines a given memory location to contain both a four byte real number **and** four individual bytes. (The same effect is obtained in FORTRAN through the use of an EQUIVALENCE statement).

The procedure is driven by the two independent random byte generators RByte1 and RByte2. As shown in Figure 8, they obtain their random bytes rather crudely by returning the first byte of a two byte integer produced by the mixed congruential generator:

$$i(k+1) = \mathbf{a} * i(k) + c \mod (32,768)$$

where : $i(k)$ = i-th integer produced by the generator
$c = 1$
$\mathbf{a} = 4 * j + 1 \quad (0 < j < 32,768)$

It can be shown [4] that each period of this generator will contain exactly one representation of **all** integers in its range. Furthermore, if **a** is carefully selected, then the resulting integers will pass most reasonable tests for randomness of distribution and sequence. Since these tests are based on the numeric value of the integer, it follows that its most significant byte of the integer will also pass these tests. It would be nice if the second byte also exhibited such properties. Unfortunately we were not able to identify a multiplier **a** (we tried them all) for which this was true. This is the reason why the second byte is discarded in the generator shown in Figure 8.

Since exactly 3 bytes are required to construct a mantissa while an average of 0.5019607 bytes are required to construct an exponent, the generators will never be synchronized. It therefore seems quite safe to use the same multiplier for both generators. This was done in the following implementation and evaluation sections.

```
function uniform :real;
type
  realtype =
    record                    {variant record for byte access}
      case integer of
        1: (unif : real);     {The variable we want}

        2: (exponent : byte; {exponent and sign bits}
            m1 : byte;        {most significant byte of mantissa}
            m2 : byte;
            m3 : byte);       {least significant byte of mantissa}
      end;
var
  k : byte;
  u : realtype;
begin
  with u do
    begin
      m1 := RByte1;       {a separate byte generator is assumed}
      m2 := RByte1;
      m3 := RByte1;
      exponent := 0;          { proper value for 0.5 < u  < 1.0    }
      if m1  < 128 then       {by convention the most significant  }
      begin                   {bit 1 in m1 must be 1               }
        m1 := m1 + 128;
        exponent := 127;
        k := RByte2;
        While k = 0 do
          begin
            exponent := exponent - 8;
            k := RByte2;
          end;
        if k < 128 then begin
          if k >= 64 then exponent := exponent -1
            else if k >= 32 then exponent := exponent -2
              else if k >= 16 then exponent := exponent - 3
                else if k >= 8 then exponent := exponent -4
                  else if k >= 4 then exponent := exponent - 5
                    else if k >= 2 then exponent := exponent -6
                      else exponent := exponent - 7
       end;
      uniform := unif;
    end;
  end;
```

Figure 7: Pascal/MT+ Implementation of uniform random number generator using AMD 9511 floating point notation.

```
function RByte1:Byte;
(  note:   the  following  declarations must appear in  the  main program
   VAR
     seed :     record
                  case integer of
                    1: (int : integer);
                    2: (lsbyte:byte;
                        msbyte:byte);
                   end;          }
  CONST
      MULT = 1221;  { other good values are: 2837, 3993, 4189, 4293,}
    begin          {                        9237,14789,15125,17245}
      seed.int := MULT * seed.int + 1;
      RandomByte := seed.msbyte;
      if seed.int < 0 then seed.in[illegible] seed.in[illegible] [illegible]nt+1;
   end;
```

Figure 8. Random byte generator for Z80 based system. Note that the order of bytes in the integer is non standard.

EVALUATION

A summary of the tests performed to assess the quality of the proposed generator is presented in Figure 9. For each test the initial seed for RByte1 was 12345 and the initial seed for RByte2 was 2345. To save space details of the test procedures are not presented here. The reader is referred to [6] or to the reference cited in Figure 9 for more information.

Property	Empirical Measure	Criteria	Procedure	Reference
Uniform Distribution	100 bin frequency count	Chi-square 99 df	1000 observations 1000 replications	[4] p.35
Random Sequence	Count of runs up of length 1,2,3,4 and 5 or more	Chi-square 5 df	1000 observations 1000 replications	[4] p.68
Lack of auto correlation	Counts of normalized acf's outside +/-2 s for all lags	No clear pattern Most counts <4	2000 observations 20 replications	[6]
Computational Effort	Time to generate 10,000 variables	Other generators	wrist watch timing on 4Mz Z80 micro	
Period	Gap between two identical sequences of 100 numbers	Other Generators	Published results used if available	

Figure 9. Summary of Evaluation tests.

The outcome of these when tests applied to the generator in Figure 7 using the byte generator in Figure 8 are shown in Figure 10.

MULT	DISTRIBUTION					RUN-TEST					AUTOCORRELATION	
	Range of Chi-square values, (1000 tests)			Test outcomes at 5% level		Range of Chi-square values (1000 obs)			Test outcomes at 5% level		Index of lags w/ too many acf's outside +/- 2 sigma	
	min	avg	max	accept	reject	min	avg	max	accept	reject	4 bad acf's	5 bad acf's
1221	61.0	99.9	155.4	932	68	0.1	4.1	20.9	979	21	109	none
2837	59.2	96.4	138.0	974	26	0.1	4.1	25.7	952	42	51	none
3993	62.6	99.0	163.2	957	43	0.1	3.8	19.1	979	21	3	none
4189	63.0	96.0	141.0	976	24	0.0	4.1	18.1	983	17	none	17
4293	57.2	95.0	138.0	976	24	0.0	3.8	20.5	985	15	41	77
9237	66.4	99.6	151.2	936	64	0.1	4.0	23.1	971	29	61	none
14789	61.2	98.6	150.4	958	42	0.1	4.2	17.9	971	29	20, 65	none
15125	69.0	98.5	146.8	979	21	0.1	4.0	14.9	977	23	1, 115	none
17245	67.2	99.5	137.2	960	40	0.1	4.0	19.4	974	26	25	42
IDEAL	66.5	98.3	139.0	50	50	0.4	4.3	16.6	950	50	2 bad lags	no bad lags

Figure 10: Results of Empirical tests for recommended seeds.

It should be noted that our subjective criteria of "closeness to Chi-square distribution" is a more discriminating test than the conventional chi-square test discussed in [4] . While the conventional test rejects the hypothesis if a single value of the test statistic is out of bound, we generate many values of the statistic and reject if the resulting empirical distribution does not appear to follow the Chi-square dis tribution. (We are able to do this since the long period of the generator allows us access to an extremely large samplesize)

In order to determine the effort required to generate a random number, we measured the time required to generate 10,000 different numbers using the present and other algorithms. All programs were run on a 4Mz Z80 based micro computer without special floating point hardware. The results of these tests are summarized in Figure 11:

Generator type	period	resolution	gap between like numbers	Time
Figure 7	at least 64,000,000	$4.7 * 10^{-10}$	random	18
Figure 7 using a Tauswortne type random byte generator [3,5,6]	at least 64,000,000	$4.7 * 10^{-10}$	random	14
Multiplicative congruential generator returning a 16 bit integer I(n+1) = I (n) * a * 1 Mod 32768	32,768	1	32,768	5
Multiplicative congruential generator returning a value on 0-1 U = I(n)*0.00003051757	32,3278	$3.1 * 10^{-5}$	32,786	18
Multiplicative congruential generator emulating 32 bit arithmetic in 16 bits I(n+1) = I(n) * a + 1 Mod 2**31	32,3278	1	32,786	133

Figure 11. Comparison between different generators.

Two implementations of the prosed generator were timed. The first [Figure 7] used the random byte generator presented in Figure 8, while the second used Figure 7 in conjunction with an experimental implementation of a Tausworthe type random byte generator [3,5,6]. The first of these was about as fast as a conventional LCG based floating point generator generator, while the second was substantially faster. Both generators yielded random number streams with statistical properties vastly superior to LCG based generators.

SUMMARY

A procedure for the construction of uniformly distributed deviates on [0-1] from two independently distributed streams of random numbers has been presented. Based on the conventions used for floating point notation, the procedure is shown to have good statistical properties. This includes the fact that the interval between two like numbers is itself a random variable (the absence of this property is frequently a reason for concern when conventional congruential generators are used). The procedure is shown to be computationally efficient, and further improvements in computational speeds should be possible through the use of a more efficient generator of random bytes. One disadvantage of the procedure is a lack of portability of the resulting computer programs. This is because (1) different machines used different conventions for representing floating point numbers, and (2) the driving byte generators are likely to be machine dependent congruential generators. However these restrictions are about the same as those experienced for conventional congruential generators.

REFERENCES

1. Intel Corp., **8231A Arithmetic Processing Unit,** Preliminary Data Sheet, 1981.

2. Intel Corp., **8232 Arithmetic Processing Unit,** Preliminary Data Sheet, 1981.

3. Lewis.T.G. and W.H.Payne, **Generalized Feedback Shift Register Pseudo Random Number Algorithm, Jounal ACM, Vol 20, No 3, July 1973, pp 456-468**

4. **Knuth,D.E.,**The Art of Computer Programming, Vol. 2, **Addison-Wesley, 1969.**

5. **Tausworte,R.C.,** Random Numbers Generated by Linear Recurrence, Modulo Two,**Math. Comput. 19 (1965) 201-209.**

6. **Thesen, A and T.J. Wang,** Some Efficient Random Number Generators for Micro Computers **ORSA/TIMS Chicago 1983.**

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 2554	2. GOVT ACCESSION NO. AD-A132 851	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* AN EFFICIENT GENERATOR OF UNIFORMLY DISTRIBUTED RANDOM DEVIATES BETWEEN ZERO AND ONE		5. TYPE OF REPORT & PERIOD COVERED Summary Report - no specific reporting period 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Arne Thesen		8. CONTRACT OR GRANT NUMBER(s) DAAG29-80-C-0041
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematics Research Center, University of Wisconsin 610 Walnut Street Madison, Wisconsin 53706		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Work Unit Number 6 - Miscellaneous Topics
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office P.O. Box 12211 Research Triangle Park, North Carolina 27709		12. REPORT DATE August 1983 13. NUMBER OF PAGES 10
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Random Number Generators, Uniform Distribution, Micro Computers

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

We show that a uniformly distributed random deviate between zero and one can be constructed by combining two independent random variables: 1) a random exponent drawn from a geometric distribution, and, 2) a random mantissa drawn from a uniform distribution. Algorithms for generating these random variables are developed, and a Pascal procedure combining these into a computationally
(cont.)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ABSTRACT (cont.)

efficient random number generator for micro computers is presented. The resulting generator is computationally efficient as it: 1) uses no floating point arithmetic, and, 2) uses on the average only 3.5 random bytes to construct a four byte random deviate. Finally we show empirically that the generator has both an unusually long period and excellent statistical properties.